Chapter 1

From Graph Grammar to STS

1.1 Requirements considerations

In order to do model-based testing with GGs, stimuli and responses have to be obtained from the GG. ATM uses an IOSTS, where the instantiated switch relations represent a stimulus to or a response from the SUT. To get an equivalent notion of stimuli/respones in GGs, the GG must be extended to an IOGG by indicating for each transformation rule whether it is of the input or output type. Then the IOGG can be explored to an IOGTS. The input/output rule transitions of the IOGTS can be used as the abstract stimuli and responses.

The second requirement for the design is the possibility to measure coverage statistics. The exploration of a GG can be done in two ways: *on the fly*, rule transitions are explored only when chosen by ATM, or *offline*, the GG is first completely explored and then sent to ATM. On-the-fly model exploration works well on large and even infinite models. However, coverage statistics cannot be calculated with this technique. The number of states (graphs) and rule transitions the model has when completely explored are not known, so a percentage cannot be derived. As coverage statistics are an important metric, the offline model exploration is chosen for GRATiS.

The last requirement is efficiency. An IOGTS can potentially be infinitely large, due to the range of data values. A model that is more efficient with data values is an STS. The setup of GRATiS is therefore to transform the IOGG directly to an IOSTS. Note that the first requirement is met, because locaton and switch relation coverage can be calculated on the IOSTS.

Taking these requirements into account, the method to achieve the goal of model-based testing on GGs is the following three steps:

1. Assign I/O types to graph transformation rules
2. Create an IOSTS from the IOGG
3. Perform the model-based testing on the IOSTS

This chapter describes an algorithm for creating an IOSTS from an IOGG.

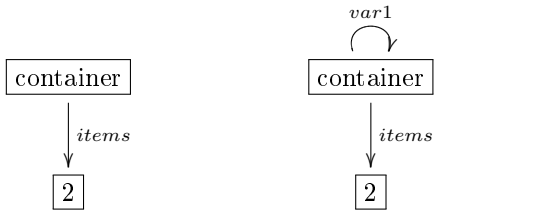
1.2 Point algebra

We define a *point algebra* $\mathcal{P}$ to be an algebra with $\forall s \in S.|\mathbb{U}^s_{\mathcal{P}}| = 1$. Each graph in $\mathcal{G}$ using the point algebra is structurally unique upto isomorphism; different values on value nodes are eliminated by the point algebra and two structurally equivalent graphs are the same graph. Therefore, using this

algebra is efficient when exploring the GTS. The loss of information is only in the concrete values at each state. This information is also not present in an STS, which treats the values symbolically as variables.

1.3 Variables

The variables in an STS represent an aspect of the modelled system. For instance, if a system keeps track of the number of items in containers, the STS modelling this system could have integer location variables $items_1..items_n$. The value nodes in a host graph are a representation of one element from the universe of elements of the same sort. Edges can exist between graph nodes and value nodes. The same example modelled in a graph grammar could be a graph node representing a container with an edge labelled 'items' to an integer node. This is shown in Figure 1.1a. This is a common way of representing a variable in a GG. Here the combination of edge plus source node represents the variable. However, the source node identity is not consistent through graph transformations, as the graphs are structurally unique upto isomorphism. In order to have variables in GGs, the source node must be made structurally unique, by means of a self-edge. Figure 1.1b shows the self-edge on the container node. The variable *var1_items*, the number of items in the container, is now represented by this graph.



(a) Common way (b) Persistent way

Figure 1.1: Possible ways of modelling variables in GGs

On the basis of the discussion above, we introduce the following terminology. The labels on the self-edges we call *variable labels*, represented by L_{var}. The edge having a variable label we call a *variable edge*. The source and target node of the variable edge we call the *variable anchor*.

1.4 The algorithm

Let $J = \langle W, w_0, \mathcal{L}, \imath, \mathcal{I}, \Lambda, D \rangle$ be an IOSTS and let $K = \langle R, G_0 \rangle$ be an IOGG. The first step in the algorithm is to explore the GG K using the point algebra $\mathcal{P}$ to an IOGTS $O_{\mathcal{P}} : \langle G, R, M, U, G_0 \rangle$.

1.4.1 Locations

The set of locations W is chosen to be equal to the set of graphs G. Additionally, the initial location w_0 is equal to the initial graph G_0.

1.4.2 Location variables

The location variables are a subset of the product of variable labels and regular labels, given by $\mathcal{L} \subset L_{var} \times L$. The set of location variables in a host graph $\langle V_{G^h}, E_{G^h} \rangle$ is defined by the following. $\langle l_{var}, l \rangle \in \mathcal{L}$ if:

- $\langle w \in \mathbb{W}, l, u \in \mathbb{U} \rangle \in E_{G^h}$ - the label must be on an edge from a graph node to a value node.
- $\langle w, l_{var}, w \rangle \in E_{G^h}$ - the variable label must be a self edge on the same graph node.

The initialization $\imath$ is then given by $\langle l_{var}, l \rangle \mapsto u$.

1.4.3 Gates

The gate of a switch relation represents the stimulus to or response from the SUT. In an IOGG, the rules are this representation. Therefore, the set of gates Λ is chosen to be equal to the set of rules R.

1.4.4 Interaction variables

Interaction variables are used by the gates to represent a stimulus or response variable. The variable nodes in rule graphs are this representation. The set of interaction variables $\mathcal{I}$ is chosen to be equal to the set of variable nodes $\mathcal{V}$. For a rule r and all variable nodes $\mathcal{V}_r$ in *LHS* of r, $arity(r) = |\mathcal{V}_r|$.

1.4.5 Guards

The guard of a switch relation restricts the use of the switch relation based on the values of the variables. In a GG, a rule is restricted by the terms. The variables used in the terms are interaction variables. Therefore, the first part of the guard is constructed by joining the terms for each term node by $\bigwedge_{z \in V_{G^r} \cap 2^{\mathcal{T}}} \bigwedge_{t_1, t_2 \in z} t_1 = t_2$. Using a rule match m, the second part is constructed. For a $LHS = \langle V_{G^r}, E_{G^r} \rangle$ the smallest set of terms T, such that $\langle m(z), l \rangle = x \in T$ when:

- $x \in \mathcal{V} \cap V_{G^r}$
- $\langle z, l, x \rangle \in E_{G^r}$
- $m(z)$ is a variable anchor

Then, the terms are joined by $\bigwedge_{t \in T} t$.

1.4.6 Update mappings

An edge with label l from a variable anchor z to a value node can be erased from the graph and a new edge with label l from z to a new value node can be created by a rule. This inidicates an update for the location variable given by $\langle z, l \rangle$. In the rule graph, the *RHS* of the rule has the pre-image of the z and the edge to a variable node, given by the interaction variable x. The update mapping for this example is: $\langle z, l \rangle \mapsto x$.

1.4.7 Switch relations

A rule transition $G \xrightarrow{r,m} G' \in U$ is mapped to a switch relation $(G \xrightarrow{r,\gamma,\rho} G') \in D$. The guard and update mapping are constructed according to sections 1.4.5 and 1.4.6 using r and m.

1.5 Constraints

This section describes the constraints on the algorithm in section 1.4.

1.5.1 Constraint 1: unique variables

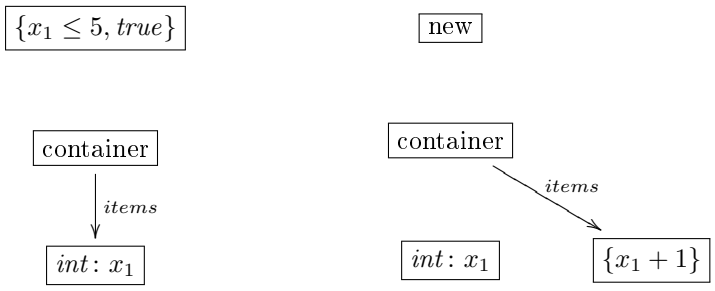
A location variable is indicated by a node and label pair $\langle z, l \rangle$. This pair must be unique, i.e. no two edges $\langle z, l, z' \rangle, \langle z, l, z'' \rangle$ may exist where $z' \neq z''$. Otherwise, it is possible that a variable has two different values.

1.5.2 Constraint 2: no variables in NACs

Let $\langle z, l, v \in \mathcal{V} \rangle$ be an edge in a rule graph in the NAC of a rule. Let $\{v, 1\}$ be a term node in the same rule graph. This is a common way of expressing that the v node may not have the value 1 as image. However, using the point algebra this rule will never match, because there is only one possible image for the variable node and the value 1 in the point algebra. A correct way of modelling this example, is having the term node $\{v = 1, false\}$ in the *LHS* of the rule. In the point algebra, both terms valuate to the same boolean value and an image for this term node can always be found.

1.5.3 Constraint 3: structural constraints on node creating rules

In the previous constraint it is shown that a term node in the *LHS* always has an image, if all terms are of the same sort. Figure 1.2 shows the *LHS* and *RHS* of a rule in the container-items example. The rule adds an item to the container unless it is full, i.e. has five items. If an item is added, a new node is created in the host graph. Using the point algebra, this rule creates an infinite number of structurally unique graphs. Therefore, the exploration never ends. Node creating rules must have structural constraint(s), such that an infinite exploration is prohibited.



(a) LHS (b) RHS

Figure 1.2: Node creating rule without structural constraint

1.6 Implementation

This section features several implementation issues of GRATiS.

1.6.1 General setup

GRATiS uses GROOVE as a replacement of the IOSTS in ATM. Figure 1.3 shows this graphically. GROOVE has several exploration strategies for exploring a GG to a GTS. GRATiS introduces two new strategies, the *remote exploration strategy* and the *symbolic exploration strategy*. The

'Exploration Strategy' is an exploration strategy in GROOVE such as the Breadth-First exploration strategy. The remote, symbolic and GROOVE exploration strategy form a chain where the possible rule transitions are passed down and the chosen rule transition is passed back up. The symbolic strategy transforms the GG to an STS based on the explored rule transitions. The remote exploration strategy waits until the IOSTS is done and then sends it to ATM. 'a' is the start of a new collaboration chain, representing the normal flow of ATM as depicted in Figure **??**.

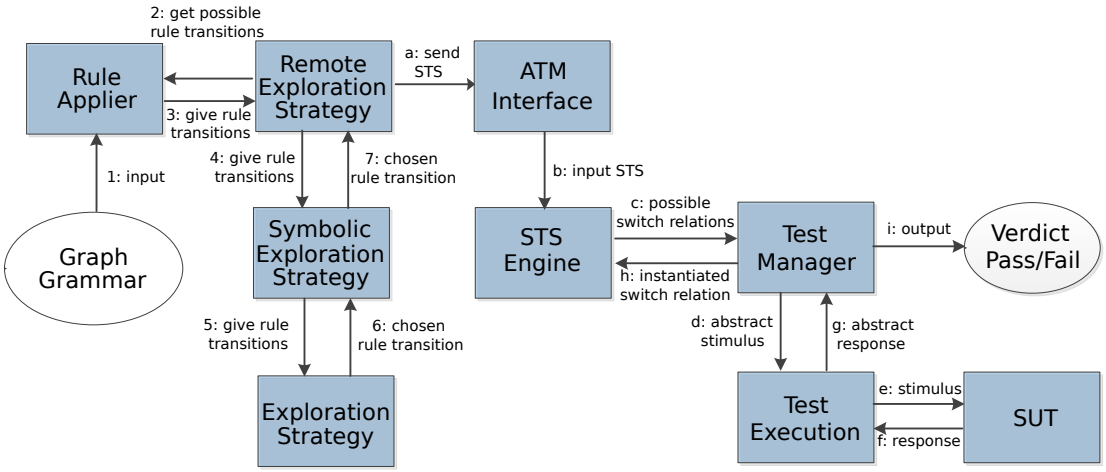


Figure 1.3: The GRATiS design: replacing the STS with GROOVE

1.6.2 Control program

GROOVE requires variable nodes to be connected to other nodes. This prevents the tool from crashing when a rule contains an unconstrained integer, real or string node. Interaction variables used in responses are often an unknown value coming from the SUT. To model this, a *control program* is needed. This program regulates which rules are applied and chooses values for variables. Figure 1.4 shows an example of a control program. Here *alap* lets the program run as long as new graphs are explored. The rule r is applied or any other rule. n is the image of a variable node marked *parin*: 0 when the rule has a match. This node does not need to be connected to other nodes. Using the point algebra, any value can be used for $n \in \mathbb{U}^s$, as long as $s \in S$ is the sort of the variable node. This example shows how a variable node can be used to model a response value from the SUT.

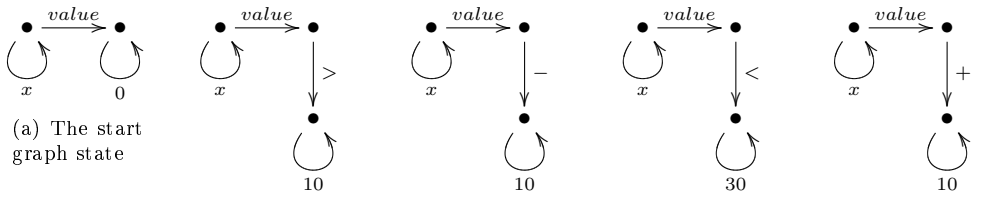
$$alap\{r(n)|other\}$$

Figure 1.4: A control program

1.6.3 Rule priority

There can be several outgoing rule transitions from a graph. However, GROOVE can set different priority levels on rules. A rule transition with a higher priority rule is explored before rule transitions with lower priority rules. Consider the graph grammar in Figure 1.5. The 'add' rule produces a rule transition to a graph, where the 'sub' rule produces a rule transition back to the start graph. The 'sub' rule does not match the start graph, because it has a lower priority than the 'add' rule.

The graphs are isomophic under the point algebra, so they represent the same location. The STS of transforming this graph grammar is in Figure 1.6, with $\imath = \{x \mapsto 25\}$. This STS is wrong,



(a) The start graph state

(b) The LHS of rule 'sub' with priority 1

(c) The RHS of rule 'sub' with priority 1

(d) The LHS of rule 'add' with priority 2

(e) The RHS of rule 'add' with priority 2

Figure 1.5: A control node and program in GROOVE

because the 'sub' switch relation can be taken from the start.

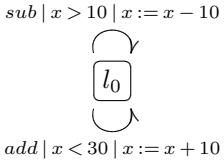


Figure 1.6: A wrong STS transformation of the graph grammar in Figure 1.5

The solution is shown in Figure 1.7. The negated guard of the 'add' switch relation is added to the 'sub' switch relation. The optimized guard for this switch relation is 'x >= 30' of course, but this shows the main principle: for each outgoing switch relation, the negated guard of all switch relations represented by higher priority rules must be added to the guard. So, the 'x < 30' guard is negated to '!(x < 30)' and added to yield the 'x > 10 && !(x < 30)' guard. Note that if the 'add' switch relation had no guard, it would be applicable on all graph states with isomorphic abstractions. Therefore, the 'sub' switch relation would not exist, because the 'add' rule is always applicable whenever the 'sub' rule also is.

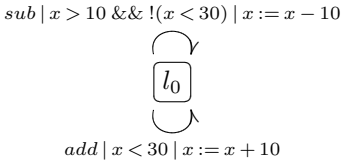


Figure 1.7: A correct STS transformation of the graph grammar in Figure 1.5